DWARF Debugging Information Format

UNIX International
Programming Languages SIG
Revision: 1.1.0 (October 6, 1992)

Published by:

UNIX International
Waterview Corporate Center
20 Waterview Boulevard
Parsippany, NJ 07054

for further information, contact:
Vice President of Marketing

Phone: +1 201-263-8400
Fax: +1 201-263-8401

International Offices:

UNIX International Asian/Pacific Office Shinei Bldg. 1F Kameido Koto-ku, Tokyo 136 Japan	UNIX International Australian Office 22/74 - 76 Monarch St. Cremorne, NSW 2090 Australia	UNIX International European Office 25, Avenue de Beaulieu 1160 Brussels Belgium	UNIX International Pacific Basin Office Cintech II 75 Science Park Drive Singapore Science Park Singapore 0511 Singapore
Phone:(81) 3-3636-1122 Fax: (81) 3-3636-1121	Phone:(61) 2-953-7838 Fax: (61) 2 953-3542	Phone:(32) 2-672-3700 Fax: (32) 2-672-4415	Phone:(65) 776-0313 Fax: (65) 776-0421

NOTICE:

UNIX International is making this documentation available as a reference point for the industry. While UNIX International believes that this specification is well defined in this second release of the document, minor changes may be made prior to products meeting this specification being made available from UNIX System Laboratories or UNIX International members.

1. INTRODUCTION

This document defines the format for the information generated by compilers, assemblers and linkage editors that is necessary for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. Instead, the goal is to create a method of communicating an accurate picture of the source program to any debugger in a form that is economically extensible to different languages while retaining backward compatibility.

The design of the debugging information format is open-ended, allowing for the addition of new debugging information to accommodate new languages or debugger capabilities while remaining compatible with other languages or different debuggers.

1.1 Purpose and Scope

The debugging information format described in this document is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by requiring language independent debugging information whenever possible. Individual needs, such as C++ virtual functions or Fortran common blocks are accommodated by creating attributes that are used only for those languages. We believe that this document sufficiently covers the debugging information needs of C, C++ and FORTRAN 77.

This document describes DWARF Version 1, which is designed to be binary compatible with the debugging information that is described in the document *DWARF Debugging Information Requirements - Issue 2*, dated April 4, 1990, and made available by AT&T to its source licensees. The April 4, 1990, document describes the debugging information that is generated by the UNIX System V Release 4 C compiler and consumed by the System V Release 4 debugger, `sdb`.

By ‘‘binary compatibility’’ we mean that

1. All features intended to support C and Fortran described in the April 4, 1990, document are included in this document, and
2. DWARF produced according to this (DWARF Version 1) specification should be considered well formed by a System V Release 4 compatible DWARF consumer, but may contain information that such a consumer is unable to interpret. Consumers are expected to ignore such information.

The intended audience for this document are the developers of both producers and consumers of debugging information, typically language compilers, debuggers and other tools that need to interpret a binary program in terms of its original source.

1.2 Overview

There are two major pieces to the description of the DWARF format in this document. The first piece is the debugging information, itself. Section two describes the overall structure of that information. Section three describes the specific debugging information entries and how they communicate the necessary information about the source program to a debugger.

The second piece of the DWARF description is the way the debugging information is encoded and represented in an object file. The DWARF encoding is presented in section four.

Section five describes the future directions of the DWARF specification.

In the following sections, text in normal font describes required aspects of the DWARF format. Text in *italics* is explanatory or supplementary material, and not part of the format definition itself.

1.3 Vendor Extensibility

This document describes only the features of DWARF that have been implemented and tested by at least one vendor (with a very few exceptions). It does not attempt to cover all languages or even to cover all of the interesting debugging information needs for its primary target languages (C, C++, Fortran). Therefore the document provides vendors a way to define their own debugging information tags, attributes, fundamental types, type modifiers, location atoms and language names, by reserving a portion of the name space and valid values for these constructs for vendor specific additions. Future versions of this document will not use names or values reserved for vendor specific additions. All names and values not reserved for vendor additions, however, are reserved for future versions of this document. See section 4 for details.

2. GENERAL DESCRIPTION

2.1 The Debugging Information Entry

DWARF uses a series of debugging information entries to define a low-level representation of a source program. Each debugging information entry consists of an identifying tag followed by a series of attributes. The tag specifies the class to which an entry belongs, and the attributes define the specific characteristics of the entry.

The set of required tag names is listed in Figure 1. The debugging information entries they identify are described in section 3.

The debugging information entries in DWARF Version 1 are intended to exist in the `.debug` section of an object file.

TAG_array_type	TAG_class_type
TAG_common_block	TAG_common_inclusion
TAG_compile_unit	TAG_entry_point
TAG_enumeration_type	TAG_formal_parameter
TAG_global_subroutine	TAG_global_variable
TAG_inheritance	TAG_inlined_subroutine
TAG_label	TAG_lexical_block
TAG_local_variable	TAG_member
TAG_module	TAG_padding
TAG_pointer_type	TAG_ptr_to_member_type
TAG_reference_type	TAG_set_type
TAG_source_file	TAG_string_type
TAG_structure_type	TAG_subrange_type
TAG_subroutine	TAG_subroutine_type
TAG_typedef	TAG_union_type
TAG_unspecified_parameters	TAG_variant
TAG_with_stmt	

Figure 1. Tag names

2.2 Attribute Types

Each attribute consists of a name/value pair. The set of attribute names is listed in Figure 2.

An attribute value can have any one of the following forms:

address	Refers to some location in the address space of the described program.
reference	Refers to some member of the set of debugging information entries that describe the program.
constant	Two, four or eight bytes of uninterpreted data.
block	An arbitrary number of uninterpreted bytes of data.
string	A null-terminated sequence of zero or more (non-null) bytes. Data in this form are generally printable strings.

There are no limitations on the ordering of attributes within a debugging information entry, but to prevent ambiguity, no more than one attribute with a given name may appear in any debugging information entry.

AT_bit_offset	AT_bit_size
AT_byte_size	AT_common_reference
AT_comp_dir	AT_const_value
AT_containing_type	AT_default_value
AT_discr	AT_discr_value
AT_element_list	AT_friends
AT_fund_type	AT_high_pc
AT_inline	AT_is_optional
AT_language	AT_location
AT_low_pc	AT_lower_bound
AT_member	AT_mod_fund_type
AT_mod_u_d_type	AT_name
AT_ordering	AT_private
AT_producer	AT_program
AT_protected	AT_prototyped
AT_public	AT_pure_virtual
AT_return_addr	AT_sibling
AT_specification	AT_start_scope
AT_stride_size	AT_string_length
AT_stmt_list	AT_subscr_data
AT_upper_bound	AT_user_def_type
AT_virtual	

Figure 2. Attribute names

2.3 Relationship of Debugging Information Entries

A variety of needs can be met by permitting a single debugging information entry to ‘‘own’’ an arbitrary number of other debugging entries and by permitting the same debugging information entry to be one of many owned by another debugging information entry. This makes it possible to describe, for example, the static block structure within a source file, show the members of a structure, union, or class, and associate declarations with source files or source files with shared objects.

The ownership of debugging information entries is represented by the physical ordering of the debugging information entries and the use of an `AT_sibling` attribute. The value of the sibling attribute of a debugging information entry is a reference to another debugging information entry. The referenced entry may be a null entry, which indicates the end of the sibling chain. All debugging information entries, except the special entry whose tag is `TAG_padding`, are required to have a sibling attribute. A debugging information entry is owned by its physical predecessor unless the same debugging information entry is referred to by the sibling attribute of that physical predecessor. If a debugging information entry is owned by its physical predecessor, it may be thought of as the ‘‘first child’’ of its predecessor. The owner of a debugging information entry is called its ‘‘parent.’’ Subsequent ‘‘children’’ of a debugging information entry derive from the first child as a chain of siblings.

Conceptually, the debugging information entries form a graph whose nodes are declarations and whose arcs either point to the next declaration owned by the same parent (sibling) or the first declaration owned by that declaration (first child). The ordering of the entries is like a traversal of the graph that always traverses the first child arc of each node before the sibling arc.

2.4 Location Information

The debugging information is required to provide a description of how to find the value of program variables and how to determine other run-time values, such as the bounds of dynamic arrays and strings.

Information on the location of program objects can be provided in a language independent fashion by creating descriptions of arbitrary complexity from a few basic building blocks, or atoms. Such descriptions are called ‘‘location descriptions’’ and form the values of `AT_location` attributes. This section describes the location atoms currently defined, how they fit together to make location descriptions, and how those descriptions are stored in the debugging information entries.

2.4.1 Location Atoms

Some of the location atoms described in this subsection can act as complete location descriptions or as parts of larger location descriptions. The remaining location atoms are operators on intermediate values and can only be used as part of a larger location description. Together, they describe a simple stack machine with postfix operations. The value on the top of the stack after ‘‘executing’’ the location description is taken to be the result (the address of the object, or the value of the array bound).

There are seven forms of location atoms:

1. A ‘‘reg(number)’’ atom indicates that the object is in the register specified by the (number).
2. A ‘‘basereg(number)’’ atom indicates that the value in the register specified by the (number) is an address to be pushed onto the stack.
3. An ‘‘addr(address)’’ atom indicates that the (address) is a relocated or relocatable address to be pushed.
4. A ‘‘const(number)’’ atom indicates that the (number) is a signed constant (unaffected by linkage editing) to be pushed.
5. A ‘‘deref2’’ atom acts as a directive to pop the stack, treat the value as an address, and push the (sign-extended, 4-byte) result of fetching two bytes from that address.
6. A ‘‘deref’’ atom acts as a directive to pop the stack, treat the value as an address, and push the data retrieved from that address. The size of the data retrieved is equivalent to the size of an address on the target machine.
7. An ‘‘add’’ atom acts as a directive to pop the top two values from the stack and push their sum.

The full list of required location atom names is given in Figure 3[1].

1. The atom name `OP_DEREF4` is reserved and is recognized as a synonym for `OP_DEREF`.

```
OP_ADD
OP_ADDR
OP_BASEREG
OP_CONST
OP_DEREF
OP_DEREF2
OP_DEREF4
OP_REG
```

Figure 3. Location atoms

2.4.2 Descriptions

A location description is one location atom or an ordered list of atoms. The complexity of the description is determined by the number of atoms in the ordered list. If location descriptions are formed from more than one location atom, those atoms are ordered as if the atoms were operators in a postfix expression. A location description containing no atoms indicates an object existing in the source code that does not exist in the executable program (possibly because of optimization).

The expression represented by a location description, if evaluated, generates the runtime address of the value of a symbol except where the reg(number) atom is used. Expressions referring to members of structures or unions expect the base address of the structure-typed object to be on the stack before being executed.

Here are some examples of how location atoms are used to form location descriptions:

`OP_REG(3)`
: *The value is in register 3.*

`OP_ADDR(0x80d0045c)`
: *The value of a static variable is at machine address 0x80d0045c.*

`OP_BASEREG(FP) OP_CONST(44) OP_ADD`
: *Add 44 to the value in the FP register to get the address of an automatic variable instance.*

`OP_BASEREG(AP) OP_CONST(32) OP_ADD OP_DEREF`
: *A call-by-reference parameter whose address is in the word 32 bytes from where the AP register points.*

`OP_CONST(4) OP_ADD`
: *A structure member is four bytes from the start of the structure instance. The base address is assumed to be already on the stack.*

The use of ''FP'' and ''AP'' in the preceding examples is meant to indicate the frame pointer *and argument pointer registers.*

2.5 Type Attributes

Program variables and other debugging information entries have attributes that describe the type of the entry (as defined by the source language). There are four type attributes: fundamental types, user-defined types, modified fundamental types and modified user-defined types.

2.5.1 Fundamental Types

A fundamental type is a data type that is not defined in terms of other data types. Each programming language has a set of fundamental types that are considered to be built into that language.

A fundamental type is represented in the debugging information as an `AT_fund_type` attribute whose value is a constant. The value corresponds to one member of the set of fundamental types whose names are enumerated in Figure 4.

FT_boolean	FT_char
FT_complex	FT_dbl_prec_complex
FT_dbl_prec_float	FT_ext_prec_complex
FT_ext_prec_float	FT_float
FT_integer	FT_label
FT_long	FT_pointer
FT_short	FT_signed_char
FT_signed_integer	FT_signed_long
FT_signed_short	FT_unsigned_char
FT_unsigned_integer	FT_unsigned_long
FT_unsigned_short	FT_void

Figure 4. Fundamental types

The type `FT_pointer` *may be used to describe the C type ‘‘void *’’. It is a more compact representation than a modified fundamental type (see below).*

The type `FT_label` *is used to describe Fortran ‘‘alternate return’’ parameters.*

2.5.2 User-Defined Type Attributes

In addition to the types that are built into a language, the user may define new data types. Some of the user-defined data types are described in their own debugging information entries. These types include structures, unions, arrays, classes and enumeration types.

There is an `AT_user_def_type` attribute whose value is a reference to the debugging information entry for a user-defined type.

2.5.3 Modified Types

There are user defined types that can be described by applying a small set of modifiers to other types. One or more modifiers may be applied to either fundamental types or user-defined types in the same fashion. There are modifiers that have the meanings ‘‘pointer to,’’ ‘‘reference to,’’ ‘‘const’’ and ‘‘volatile.’’

The ‘‘pointer to’’ modifier means that the value is the address of the data of the type modified. The ‘‘reference to’’ modifier means that the value is a C++ reference to data of the type modified. The ‘‘const’’ modifier means that the variable has been qualified by the ANSI C or

C++ qualifier `const`*. The ''volatile'' modifier means that the variable has been qualified by the ANSI C qualifier* `volatile`*.*

The full set of type modifier names are listed in figure 5.

```
MOD_const
MOD_pointer_to
MOD_reference_to
MOD_volatile
```

Figure 5. Type modifiers

Modifiers are applied by prefixing a fundamental type value or a user-defined type reference with one or more modifiers. The modifiers appear as if part of a right-associative expression involving the fundamental or user-defined type. When one or more modifiers are applied to a fundamental type, the modifiers and the fundamental type value are stored in a block of contiguous bytes that are the value of the `AT_mod_fund_type` attribute. When one or more modifiers are applied to a user-defined type, the modifiers and the user-defined type reference are stored in a block of contiguous bytes that are the value of the `AT_mod_u_d_type` attribute.

As examples of how type modifiers are ordered, take the following C declarations:

`const char * volatile p;`

which represents a volatile pointer to a constant character.
This is encoded in DWARF as:
`MOD_volatile MOD_pointer_to MOD_const FT_char`

`volatile char * const p;`

on the other hand, represents a constant pointer
to a volatile character.
This is encoded as:
`MOD_const MOD_pointer_to MOD_volatile FT_char`

2.5.4 Accessibility Attributes

Some languages, notably C++ and Ada, have the concept of the accessibility of an object or of some other program entity. The accessibility specifies which classes of other program objects are permitted access to the object in question.

There are three accessibility attributes currently defined: `AT_private`, `AT_protected` and `AT_public`. The value of each of these attributes is a string containing only the terminating null byte. The presence alone of the attribute indicates the accessibility of the containing debugging entry.

3. DEBUGGING INFORMATION ENTRIES

This section describes the debugging information entries currently defined and the attributes they contain.

3.1 Compilation Unit Entries

An object file may be derived from one or more compilation units. Each such compilation unit will be described by a debugging information entry with the tag `TAG_compile_unit`.[2]

A compilation unit typically represents the text and data contributed to an executable by a single relocatable object file. It may be derived from several source files, including pre-processed ''include files.''

The compilation unit entry may have the following attributes:

1. A sibling attribute whose value is a reference to the debugging information entry loaded immediately after the last debugging information entry for that compilation unit.

   There may not actually be a debugging information entry at the indicated offset. If that offset is greater than or equal to the size of the `.debug` *section, then that offset is beyond the last valid debugging information entry.*

   As mentioned in section 2.3, all debugging information entries except entries with the special tag `TAG_padding` *have sibling attributes. The descriptions of other debugging information entries will not mention this attribute explicitly.*

2. An `AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for that compilation unit.

3. An `AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for that compilation unit.

   The address may be beyond the last valid instruction in the executable, of course, for this and other similar attributes.

4. An `AT_name` attribute whose value is a null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.

5. An `AT_language` attribute whose constant value is a code indicating the source language of the compilation unit. The set of language names and their meanings are given in Figure 6.

6. An `AT_stmt_list` attribute whose value is a reference to a table of line number information.

   This table currently exists in a separate object file section from the debugging information entries themselves. The value of the statement list attribute is the offset in the `.line` section of the first byte of the table for that compilation unit. See section 3.11.

7. An `AT_comp_dir` attribute whose value is a null-terminated string containing the current working directory of the compilation command that produced this compilation unit in

2. The tag name `TAG_source_file` is reserved and is recognized as a synonym for `TAG_compile_unit`.

LANG_ADA83	ADA
LANG_C	Non-ANSI C, such as K&R
LANG_C89	ISO/ANSI C
LANG_C_PLUS_PLUS	C++
LANG_COBOL74	ANSI COBOL-74
LANG_COBOL85	ANSI COBOL-85
LANG_FORTRAN77	FORTRAN77
LANG_FORTRAN90	Fortran90
LANG_MODULA2	Modula2
LANG_PASCAL83	ISO/ANSI Pascal

Figure 6. Language names

whatever form makes sense for the host system.

The suggested form for the value of the `AT_comp_dir` *attribute on* UNIX *systems is ''hostname*:*pathname''. If no hostname is available, the suggested form is ''*:*pathname''.*

8. An `AT_producer` attribute whose value is a null-terminated string containing information about the compiler that produced the compilation unit. The actual contents of the string will be specific to each producer, but should begin with the name of the compiler vendor or some other identifying character sequence that should avoid confusion with other producer values.

A compilation unit entry owns debugging information entries that represent the declarations made in the corresponding compilation unit.

3.2 Modules

Several languages have the concept of a ''module.'' A module is represented by a debugging information entry with the tag `TAG_module`. Module entries may own other debugging information entries describing program entities whose declaration scopes end at the end of the module itself.

If the module has a name, the module entry has a name attribute whose value is a null-terminated string containing the module name as it appears in the source program.

If the module contains initialization code, the module entry has a low pc attribute whose value is the relocated address of the first machine instruction generated for that initialization code. It also has a high pc attribute whose value is the relocated address of the first location past the last machine instruction generated for the initialization code.

3.3 Subroutine and Entry Point Entries

The following tags exist to describe debugging information entries for subroutines and entry points:

`TAG_global_subroutine`	A global subroutine or function.
`TAG_subroutine`	A file static subroutine or function.
`TAG_inlined_subroutine`	A particular inlined instance of a subroutine or function.
`TAG_entry_point`	A Fortran entry point.

3.3.1 General Subroutine and Entry Point Information

The subroutine or entry point entry has a name attribute whose value is a null-terminated string containing the subroutine or entry point name as it appears in the source program.

Note that since the names of subroutines (and other program objects described by DWARF) are the names as they appear in the source program, implementations of language translators that use some form of mangled name (as do many implementations of C++) should use the unmangled form of the name in the DWARF `AT_name` *attribute, including the keyword* `operator`*, if present. Sequences of multiple whitespace characters may be compressed.*

The subroutine entry for a member function definition for a member function defined outside of a class, structure or union body has an `AT_member` attribute whose value is a reference to a type definition of the class or structure.

Additional attributes for member functions are described in section 3.8.4.3.

The presence of the member attribute implies that the subroutine is a member of the type specified by the member attribute. The member attribute makes C++ identifier resolution through member functions possible.

If the semantics of the language of the compilation unit containing the subroutine entry distinguishes between ordinary subroutines and subroutines that can serve as the ‘‘main program,’’ that is, subroutines that cannot be called directly following the ordinary calling conventions, then the debugging information entry for such a subroutine may have an `AT_program` attribute, whose value is a string consisting of only the terminating null byte. The presence alone of this attribute marks the subroutine entry as a main program.

The program attribute is intended to support Fortran main programs. It is not intended as a way of finding the entry address for the program.

3.3.2 Subroutine and Entry Point Return Types

If the subroutine or entry point is a function that returns a value, then its debugging information entry has one of the four type attributes (fundamental type, modified fundamental type, user-defined type or modified user-defined type) described in section 2.5, to denote the type returned by that function.

Debugging information entries for C `void` *functions should not have an attribute for the return type.*

In ANSI-C there is a difference between the types of functions declared using function prototype style declarations and those declared using non-prototype declarations.

A subroutine entry declared with a function prototype style declaration may have an `AT_prototyped` attribute, whose value is a string consisting of only the terminating null byte. The presence alone of this attribute marks the subroutine entry as prototyped.

3.3.3 Subroutine and Entry Point Locations

A subroutine entry has a low pc attribute whose value is the relocated address of the first machine instruction generated for the subroutine. It also has a high pc attribute whose value is the relocated address of the first location past the last machine instruction generated for the subroutine.

Note that for the low and high pc attributes to have meaning, DWARF makes the assumption that the code for a single subroutine is allocated in a single contiguous block of memory.

An entry point has a low pc attribute whose value is the relocated address of the first machine instruction generated for the entry point.

3.3.4 Declarations Owned by Subroutines and Entry Points

The declarations enclosed by a subroutine or entry point are represented by debugging information entries that are owned by the subroutine or entry point entry. Entries representing the formal parameters of the subroutine or entry point appear in the same order as the corresponding declarations in the source program.

There is no ordering requirement on entries for declarations that are children of subroutine or entry point entries but that do not represent formal parameters. The formal parameter entries may be interspersed with other entries used by formal parameter entries, such as type entries.

The unspecified parameters of a variable parameter list are represented by a debugging information entry with the tag `TAG_unspecified_parameters`.

The entry for a subroutine or entry point that includes a Fortran common block has a child entry with the tag `TAG_common_inclusion`. The common inclusion entry has an `AT_common_reference` attribute whose value is a reference to the debugging entry for the common block being included (see section 3.7).

3.3.5 Low-Level Information

A subroutine or entry point entry may have an `AT_return_addr` attribute, whose value is a location description. The location calculated is the place where the return address for the subroutine or entry point is stored.

3.3.6 Inlined Subroutines

The representation of inlined subroutines has two pieces, the original declaration of the subroutine and each inlined instance. The declaration or ‘‘out of line’’ instance of the subroutine, if one has been generated, is represented by a subroutine or global subroutine debugging information entry. It does not describe any one particular call site for that subroutine. Such entries have an `AT_inline` attribute, whose value is a string consisting of only the terminating null byte. The presence alone of this attribute marks the subroutine entry as inlined. If no out of line instance has been generated for the subroutine, then the subroutine entry will have no low or high pc attributes.

Note that if such a subroutine entry describing only the abstract declaration of an inlined subroutine has children describing the parameters to the subroutine, those children will not have location descriptions.

Each inlined instance of such a subroutine will have a debugging entry with the tag `TAG_inlined_subroutine`. This entry has an `AT_specification` attribute whose value is a reference to the debugging entry representing the declaration or out of line instance of the subroutine. Each inlined subroutine entry owns its own copies of entries describing the parameters to that subroutine, its local variables and declarations for named types.

3.4 Lexical Block Entries

A lexical block is a bracketed sequence of source statements that may contain any number of declarations. In some languages (C and C++) blocks can be nested within other blocks to any *depth.*

A lexical block is represented by a debugging information entry with the tag `TAG_lexical_block`.

The lexical block entry has a low pc attribute whose value is the relocated address of the first machine instruction generated for the lexical block. The lexical block entry also has a high pc attribute whose value is the relocated address of the first location past the last machine instruction generated for the lexical block.

If a name has been given to the lexical block in the source program, then the corresponding lexical block entry has a name attribute whose value is a null-terminated string containing the name of the lexical block as it appears in the source program.

This is not the same as a C or C++ label (see below).

The lexical block entry owns debugging information entries that describe the declarations within that lexical block. There is one such debugging information entry for each local declaration of an identifier or inner lexical block.

3.5 Label Entries

A label is a way of identifying a source statement. A labeled statement is usually the target of one or more ‘‘go to’’ statements.

A label is represented by a debugging information entry with the tag `TAG_label`. The entry for a label should be owned by the debugging information entry representing the scope within which the name of the label could be legally referenced within the source program.

The label entry has a low pc attribute whose value is the relocated address of the first machine instruction generated for the first executable statement immediately following the label in the source program. The label entry also has a name attribute whose value is a null-terminated string containing the name of the label as it appears in the source program.

3.6 Program Variable Entries

Global variables, formal parameters and local variables are represented by debugging information entries with the tags `TAG_global_variable`, `TAG_formal_parameter` and `TAG_local_variable`, respectively.

The local variable tag is also used for file static variables in C and C++.

The debugging information entry for a program variable may have the following attributes:

1. A name attribute whose value is a null-terminated string containing the variable name as it appears in the source program.
2. An `AT_location` attribute, whose value describes the location of the variable.

   If no location attribute is present, or if the location attribute is present but describes a null entry (as described in section 2.4), the variable is assumed to exist in the source code but not in the executable program (but see number 7, below).
3. One of the four type attributes (fundamental type, modified fundamental type, user-defined type or modified user defined type).
4. If the variable entry represents the defining declaration for a C++ static data member of a structure class or union, the entry has an `AT_member` attribute, whose value is a reference

to the structure, class or union type of which this object is a member.

5. If the variable represents a Fortran optional parameter, it has an `AT_is_optional` attribute, whose value is a string consisting of only the terminating null byte. The presence alone of this attribute marks the parameter as optional.

6. A formal parameter entry describing a formal parameter that has a default value may have an `AT_default_value` attribute. The value of this attribute may be the address of a function within the program which, when called with no actual arguments, yields a value representing the default value of the parameter and whose type is the same as that of the parameter. Alternatively, the value of this attribute may be a string or any of the constant data or data block forms, in which case the value represents the actual constant default value of the parameter as represented on the target architecture.

7. A variable entry describing a variable whose value is constant and not represented by an object in the address space of the program does not have a location attribute. Such an entry has an `AT_const_value` attribute, whose value may be a string or any of the constant data or data block forms, as appropriate for the representation of the variable's value. The value of this attribute is the actual constant value of the variable, represented as it would be on the target architecture.

8. If the scope of the variable begins sometime after the low pc value for the scope most closely enclosing the variable, the variable may have an `AT_start_scope` attribute. The value of this attribute is the offset of the beginning of the scope for the variable from the low pc value of the debugging information entry that defines its scope.

   The scope of a variable may begin somewhere in the middle of a lexical block in a language that allows executable code in a block before a variable declaration, or where one declaration containing initialization code may change the scope of a subsequent declaration. For example, in the following C code:

   ```
   float x = 99.99;

   int myfunc()
   {
                   float f = x;
                   float x = 88.99;

                   return 0;
   }
   ```

   ANSI-C scoping rules require that the value of the variable `x` *assigned to the variable* `f` *in the initialization sequence is the value of the global variable* `x`*, rather than the local* `x`*, because the scope of the local variable* `x` *only starts after the full declarator for the local* `x`.

3.7 Common Block Entries

A Fortran common block may be described by a debugging information entry with the tag `TAG_common_block`. The common block entry has a name attribute whose value is a null-terminated string containing the common block name as it appears in the source program. It also has an `AT_location` attribute whose value describes the location of the beginning of the common block. The common block entry owns debugging information entries describing the variables contained within the common block.

3.8 User-Defined Type Entries

There are several debugging information entry types that describe user-defined data types, including typedefs, pointers, references, arrays, structures, unions, classes, enumerations, subroutines, strings, sets and subranges.

If the scope of the declaration of a named type begins sometime after the low pc value for the scope most closely enclosing the declaration, the declaration may have an `AT_start_scope` attribute. The value of this attribute is the offset of the beginning of the scope for the declaration from the low pc value of the debugging information entry that defines its scope.

3.8.1 Typedef Entries

Any arbitrary type named via a typedef is represented by a debugging information entry with the tag `TAG_typedef`. The typedef entry has a name attribute whose value is a null-terminated string containing the name of the typedef as it appears in the source program. The typedef entry also contains one of the four type attributes (fundamental type, user defined type, modified fundamental type, or modified user defined type).

3.8.2 Pointer and Reference Type Entries

Several languages share the concept of a ''pointer,'' which is an object whose value is the address of another object. A pointer can also be ''null,'' which means that it does not refer to any entity. In addition to the pointer, C++ supports the concept of a ''reference.'' A reference is a fixed pointer to another object. It may be thought of as an alternate name for the object with which it has been initialized or as a pointer that is automatically de-referenced.

A pointer type may be represented by a debugging information entry with the tag `TAG_pointer_type`. A reference type may be represented by a debugging information entry with the tag `TAG_reference_type`.

If a name has been given to the pointer or reference type in the source program, then the corresponding pointer type entry or reference type entry has a name attribute whose value is a null-terminated string containing the pointer type name or reference type name as it appears in the source program.

The pointer type entry or reference type entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined type attribute to denote the type pointed to or referenced.

3.8.3 Array Type Entries

Many languages share the concept of an ''array,'' which is a table of components of identical type.

An array type is represented by a debugging information entry with the tag `TAG_array_type`.

If a name has been given to the array type in the source program, then the corresponding array type entry has a name attribute whose value is a null-terminated string containing the array type name as it appears in the source program.

The array type entry describing a multidimensional array may have an `AT_ordering` attribute whose constant value is interpreted to mean either row-major or column-major ordering of array elements. The required attribute names are listed in Figure 7. If no ordering attribute is present, the default ordering for the source language (which is indicated by the `AT_language` attribute of the enclosing compilation unit entry) is assumed.

```
ORD_col_major
ORD_row_major
```

Figure 7. Array ordering

The ordering attribute may optionally appear on one-dimensional arrays; it will be ignored.

The array subscripts and the data type of the elements of the array are described by a subscript data attribute (`AT_subscr_data`) whose value is stored in a block of contiguous bytes. The subscript data attribute consists of a list of data items. There is a data item describing each array dimension and an item describing the element type. The data items describing the array dimensions are ordered to reflect the appearance of the dimensions in the source program (i.e. leftmost dimension first, next to leftmost dimension second, and so on). The last data item in the subscript data attribute is the description of the element type.

Each data item describing an array dimension consists of four parts, ordered as follows:

1. A format specifier describing the information that follows.
2. The type of the subscript index. This may be either a fundamental type value or a user-defined type reference. In either case, there is no attribute tag, but the value has the same form as a fundamental type or user-defined type reference when used as an attribute value.
3. Information describing the low bound of the array dimension. This may be either a signed constant or a location description. The location description is contained in a block of contiguous bytes; its generated value is the address of the lowest numbered element of this array dimension calculated during the execution of a program using that array. An unspecified lower bound is represented by a block of contiguous bytes with a length of zero. As with the subscript types, neither the constant nor the location description has a preceding attribute tag, but each follows the form for constant or location description attribute values.
4. Information describing the high bound of the array dimension. Again, this may either be a signed constant or a location description. An unspecified upper bound is represented by a block of contiguous bytes with a length of zero.

The data item for the element type is constructed from a format specifier followed by either a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute or a modified user defined type attribute.

The format specifier determines how the pieces of the data items should be interpreted and replaces the need for specific attribute tags to describe subscript index types and upper and lower dimension bounds. The nine possible values of the format specifier byte have the following names and meanings:

`FMT_FT_C_C`	A fundamental type followed by a constant followed by a constant.
`FMT_FT_C_X`	A fundamental type followed by a constant followed by a location description.
`FMT_FT_X_C`	A fundamental type followed by a location description followed by a constant.
`FMT_FT_X_X`	A fundamental type followed by a location description followed by a location description.
`FMT_UT_C_C`	A user-defined type reference followed by a constant followed by a constant.
`FMT_UT_C_X`	A user-defined type reference followed by a constant followed by a location description.
`FMT_UT_X_C`	A user-defined type reference followed by a location description followed by a constant.
`FMT_UT_X_X`	A user-defined type reference followed by a location description followed by a location description.
`FMT_ET`	A type attribute for the element type.

Note that the order of components of a subscript data item is fixed and independent of the value of its format specifier.

If the amount of storage allocated to hold each element of an object of the given array type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the array type entry has an `AT_stride_size` attribute, whose constant value represents the size in bits of each element of the array.

If the size of the entire array can be determined statically at compile time, the array type entry may have an `AT_byte_size` attribute, whose constant value represents the total size in bytes of an instance of the array type.

Note that if the size of the array can be determined statically at compile time, this value can usually be computed by multiplying the number of array elements by the size of each element.

In languages, such as ANSI-C, in which there is no concept of a ‘‘multidimensional array,’’ an array of arrays may be represented by a debugging information entry for a multidimensional array.

3.8.4 Structure, Union, and Class Type Entries

The languages C, C++, and Pascal, among others, allow the programmer to define types that are collections of related components. In C and C++, these collections are called ‘‘structures.’’ In Pascal, they are called ‘‘records.’’ The components may be of different types. The components are called ‘‘members’’ in C and C++, and ‘‘fields’’ in Pascal.

The components of these collections each exist in their own space in computer memory. The components of a C or C++ ‘‘union’’ all coexist in the same memory.

Pascal and other languages have a ‘‘discriminated union,’’ also called a ‘‘variant record.’’ Here, selection of a number of alternative substructures (‘‘variants’’) is based on the value of a component that is not part of any of those substructures (the ‘‘discriminant’’).

Among the languages discussed in this document, the ''class'' concept is unique to C++. A class is similar to a structure. A C++ class or structure may have ''member functions'' which are subroutines that are within the scope of a class or structure.

3.8.4.1 General Structure Description

Structure, union, and class types are represented by debugging information entries with the tags `TAG_structure_type`, `TAG_union_type` and `TAG_class_type`, respectively. If a name has been given to the structure, union, or class in the source program, then the corresponding structure type, union type, or class type entry has a name attribute whose value is a null-terminated string containing the type name as it appears in the source program.

If the size of an instance of the structure type, union type, or class type entry can be determined statically at compile time, the entry has a byte size attribute whose constant value is the number of bytes required to hold an instance of the structure, union, or class, and any padding bytes.

For C and C++, a structure, union or class entry that does not have a byte size attribute represents the declaration of an incomplete structure, union or class type.

The members of a structure, union, or class are represented by debugging information entries that are owned by the corresponding structure type, union type, or class type entry and appear in the same order as the corresponding declarations in the source program.

Data member declarations occurring within the declaration of a structure, union or class type are considered to be ''definitions'' of those members, with the exception of C++ ''static'' data members, whose definitions appear outside of the declaration of the enclosing structure, union or class type. Function member declarations appearing within a structure, union or class type declaration are definitions only if the body of the function also appears within the type declaration.

If the definition for a given member of the structure, union or class does not appear within the body of the declaration, that member also has a debugging information entry describing its definition. That entry will have an `AT_member` attribute referencing the structure, union or class declaration containing the given member. The debugging entry owned by the body of the structure, union or class debugging entry and representing a non-defining declaration of the data or function member will not have information about the location of that member (low and high pc attributes for function members, location descriptions for data members).

3.8.4.1.1 Derived Classes and Structures

The class type or structure type entry that describes a derived class or structure owns debugging information entries describing each of the classes or structures it is derived from, ordered as they were in the source program. Each such entry has the tag `TAG_inheritance`.

An inheritance entry has a user-defined type attribute whose value is a reference to the debugging information entry describing the structure or class from which the parent structure or class of the inheritance entry is derived. It also has a location attribute describing the location of the beginning of the data members contributed to the entire class by this subobject relative to the beginning address of the data members of the entire class.

An inheritance entry may have one of the three accessibility attributes (`AT_public`, `AT_private` or `AT_protected`). If no accessibility attribute is present, `AT_private` is assumed. If the structure or class referenced by the inheritance entry serves as a virtual base class, the inheritance entry has an `AT_virtual` attribute, whose value is a string consisting only of the terminating null byte. The presence alone of this attribute marks the base structure or class as virtual.

3.8.4.1.2 Friends

If the declaration of a structure, union or class type specifies ‘‘friends’’ to that structure, union or class, then the debugging entry for the type may have an `AT_friends` attribute. The value of this attribute is a list of references to the debugging information entries for the structures, unions, classes or functions declared to be friends to the structure, union or class containing the friends attribute.

3.8.4.2 Structure Data Member Entries

A data member (as opposed to a member function) is represented by a debugging information entry with the tag `TAG_member.` The member entry for a named member has a name attribute whose value is a null-terminated string containing the member name as it appears in the source program.

The structure data member entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined type attribute to denote the type of that member.

If the member entry is defined in the structure or class body, it has an `AT_location` attribute whose value is a location description that describes the location of that member relative to the base address of the structure, union, or class that most closely encloses the corresponding member declaration.

The addressing expression represented by the location description for a structure data member expects the base address of the structure data member to be on the expression stack before being evaluated.

If the member entry describes a bit field, then that entry has the following attributes:

1. An `AT_byte_size` attribute whose constant value is the number of bytes that contain an instance of the bit field and any padding bits.
2. An `AT_bit_offset` attribute whose constant value is the number of bits to the left of the leftmost (most significant) bit of the bit field value.
3. An `AT_bit_size` attribute whose constant value is the number of bits occupied by the bit field value.

The location description for a bit field calculates the address of an anonymous object containing the bit field. The address is relative to the structure, union, or class that most closely encloses the bit field declaration. The number of bytes in this anonymous object is the value of the byte size attribute of the bit field. The offset (in bits) from the most significant bit of the anonymous object to the most significant bit of the bit field is the value of the bit offset attribute.

For example, take one possible representation of the following structure definition in both big and little endian byte orders:

```
struct S {
                int   j:5;
                int   k:6;
                int   m:5;
                int   n:8;
};
```

In both cases, the location descriptions for the debugging information entries for `j`, `k`, `m` *and* `n` *describe the address of the same 32-bit word that contains all three members. (In the big-endian case, the location description addresses the most significant byte, in the little-endian case, the least significant). The following diagram shows the structure layout and lists the bit offsets for each case. The offsets are from the most significant bit of the object addressed by the location description.*

Bit Offsets:
`j:0`
`k:5`
`m:11`
`n:16`

Big-Endian

0 `j` 31	`k` 26	`m` 20	`n` 15	*pad* 7 0

Bit Offsets:
`j:27`
`k:21`
`m:16`
`n:8`

Little-Endian

pad 31	`n` 23	`m` 15	`k` 10	`j` 4 0 0

3.8.4.3 Structure Member Function Entries

A member function is represented in the debugging information by a debugging information entry with the tag `TAG_global_subroutine`. The member function entry may contain the same attributes and follows the same rules as non-member global subroutine entries (see section 3.3).

If the member function entry describes a virtual function, then that entry has either an `AT_virtual` attribute, or an `AT_pure_virtual` attribute, either of whose values is a string consisting only of the terminating null byte. The presence alone of these attributes marks the member function as virtual or pure virtual.

An entry for a virtual function also has a location attribute whose value contains a location description yielding the address of the slot for the function within the virtual function table for the enclosing class or structure.

3.8.4.4 Variant Entries

The variants of a discriminated union are represented by debugging information entries that are owned by the corresponding structure type entry. These entries appear in the same order as the corresponding declarations in the same program. A variant is represented by a debugging information entry with the tag `TAG_variant`.

The variant entry has an `AT_discr` attribute whose value is a reference to the member declaration whose value determines the variant. The variant entry also has an `AT_discr_value` attribute whose value is the value of the discriminant that applies to the variant.

The components based on a variant are represented by debugging information entries owned by the corresponding variant entry and appear in the same order as the corresponding declarations in the source program.

3.8.5 Enumeration Type Entries

An ‘‘enumeration type’’ is a scalar that can assume one of a fixed number of symbolic values.

An enumeration type is represented by a debugging information entry with the tag `TAG_enumeration_type`.

If a name has been given to the enumeration type in the source program, then the corresponding enumeration type entry has a name attribute whose value is a null-terminated string containing the enumeration type name as it appears in the source program. These entries also have a byte size attribute whose constant value is the number of bytes required to hold an instance of the enumeration.

Information about the enumeration literals is stored in an `AT_element_list` attribute whose value is a list of data elements stored in a block of contiguous bytes. Each data item consists of a constant followed by a null-terminated block of contiguous bytes. The constant will contain the internal value of the enumeration element. The null-terminated block of bytes holds the enumeration literal string as it appears in the source program. The data elements in an element list should be generated in the reverse order to the order in which they appear in the source program.

For consistency, DWARF specifies an ordering for enumeration entries. Reverse order was chosen for compatibility with existing implementations.

Examples of structure and enumeration descriptions appear in Appendix 2.

3.8.6 Subroutine Type Entries

It is possible in C to declare pointers to subroutines that return a value of a specific type. In both ANSI C and C++, it is possible to declare pointers to subroutines that not only return a value of a specific type, but accept only arguments of specific types. The type of such pointers would be described with a ‘‘pointer to’’ modifier applied to a user-defined type.

A subroutine type is represented by a debugging information entry with the tag `TAG_subroutine_type`. If a name has been given to the subroutine type in the source program, then the corresponding subroutine type entry has a name attribute whose value is a null-terminated string containing the subroutine type name as it appears in the source program.

If the subroutine type describes a function that returns a value, then the subroutine type entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user-defined type attribute to denote the type returned by the subroutine. If the types of the arguments are necessary to describe the subroutine type, then the corresponding subroutine type entry owns debugging information entries that describe the arguments. These debugging information entries appear in the order that the corresponding argument types appear in the source program.

In ANSI-C there is a difference between the types of functions declared using function prototype style declarations and those declared using non-prototype declarations.

A subroutine entry declared with a function prototype style declaration may have an `AT_prototyped` attribute, whose value is a string consisting of only the terminating null byte. The presence alone of this attribute marks the subroutine entry as prototyped.

Each debugging information entry owned by a subroutine type entry has a tag whose value has one of two possible interpretations.

1. Each debugging information entry that is owned by a subroutine type entry and that defines a single argument of a specific type has the tag `TAG_formal_parameter`.

   The formal parameter entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined type attribute to denote the type of the corresponding formal parameter.

2. The unspecified parameters of a variable parameter list are represented by a debugging information entry owned by the subroutine type entry with the tag `TAG_unspecified_parameters`.

3.8.7 String Type Entries

A ''string'' is a sequence of characters that have specific semantics and operations that separate them from arrays of characters. Fortran is one of the languages that has a string type.

A string type is represented by a debugging information entry with the tag `TAG_string_type`. If a name has been given to the string type in the source program, then the corresponding string type entry has a name attribute whose value is a null-terminated string containing the string type name as it appears in the source program.

The string type entry may have an `AT_string_length` attribute whose value is stored in a block of contiguous bytes. The value of the string length attribute is a location description yielding the location where the length of the string is stored in the program. The string type entry may also have a byte size attribute, whose constant value is the size in bytes of the data to be retrieved from the location referenced by the string length attribute. If no byte size attribute is present, the size of the the data to be retrieved is the same as the size of the fundamental type `FT_long` on the target machine.

If no string length attribute is present, the string type entry may have a byte size attribute, whose constant value is the length in bytes of the string.

3.8.8 Set Entries

Pascal provides the concept of a ''set,'' which represents a group of values of ordinal type.

A set is represented by a debugging information entry with the tag `TAG_set_type`. If a name has been given to the set type, then the set type entry has a name attribute whose value is a null-terminated string containing the set type name as it appears in the source program.

The set type entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined type attribute to denote the type of an element of the set.

If the amount of storage allocated to hold each element of an object of the given set type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the set type entry has a byte-size attribute, whose constant value represents the size in bytes of an instance of the set type.

3.8.9 Subrange Types

Several languages support the concept of a ''subrange'' type object. These objects can represent a subset of the values that an object of the basis type for the subrange can represent.

A subrange type is represented by a debugging information entry with the tag `TAG_subrange_type`. If a name has been given to the subrange type, then the subrange type entry has a name attribute whose value is a null-terminated string containing the subrange type name as it appears in the source program.

The subrange entry may have one of the four type attributes (fundamental type, modified fundamental type, user-defined type, modified user-defined type) to describe the type of object of whose values this subrange is a subset.

If the amount of storage allocated to hold each element of an object of the given subrange type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the subrange type entry has a byte-size attribute, whose constant value represents the size in bytes of each element of the subrange type.

The subrange entry may have the attributes `AT_lower_bound` and `AT_upper_bound` to describe, respectively, the lower and upper bound values of the subrange. If a bound value is described by a constant not represented in the program's address space that can be represented by one of the constant attribute forms, then the value of the lower or upper bound attribute may be one of the constant types. Otherwise, the value of the lower or upper bound attribute is a reference to a debugging information describing an object containing the bound value or itself describing a constant value.

If either the lower or upper bound values are missing, the bound value is assumed to be a language-dependent default constant.

The default lower bound value for C or C++ is 0. For Fortran, it is 1. No other default values are currently defined by DWARF.

If the subrange entry has no type attribute describing the basis type, the basis type is assumed to be the same as the object described by the lower bound attribute (if it references an object). If there is no lower bound attribute, or it does not reference an object, the basis type is the type of the upper bound attribute (if it references an object). If there is no upper bound attribute or it does not reference an object, the type is assumed to be the same type object represented by the fundamental type `FT_long`.

3.8.10 Pointer to Member Types

In C++, a pointer to a data or function member of a class or structure is a unique type.

A debugging information entry representing the type of an object that is a pointer to a structure or class member has the tag `TAG_ptr_to_member_type`.

If the pointer to member type has a name, the pointer to member entry has a name attribute, whose value is a null-terminated string containing the type name as it appears in the source program.

The pointer to member entry has one of the four type attributes (fundamental type, modified fundamental type, user-defined type, modified user-defined type) to describe the type of the class or structure member to which objects of this type may point.

The pointer to member entry also has an `AT_containing_type` attribute, whose value is a reference to a debugging information entry for the class or structure to whose members objects of this type may point.

3.9 With Statement Entries

Both Pascal and Modula support the concept of a ''with'' statement. The with statement specifies a sequence of executable statements within which the fields of a record variable may be referenced, unqualified by the name of the record variable.

A with statement is represented by a debugging information entry with the tag `TAG_with_stmt`. A with statement entry has a low pc attribute whose value is the relocated address of the first machine instruction generated for the body of the with statement. A with statement entry also has a high pc attribute whose value is the relocated address of the first location after the last machine instruction generated for the body of the statement.

The with statement entry has a user-defined type attribute, denoting the type of record whose fields may be referenced without full qualification within the body of the statement. It also has a location description attribute, describing how to find the base address of the record object referenced within the body of the with statement.

3.10 Accelerated Access

A debugger frequently needs to find the debugging information for a program object defined outside of the compilation unit where the debugged program is currently stopped. Sometimes it will know only the name of the object; sometimes only the address. To find the debugging information associated with a global object by name, using the DWARF debugging information entries alone, a debugger would need to run through all entries at the highest scope within each compilation unit. For lookup by address, for a subroutine, a debugger can use the low and high pc attributes of the compilation unit entries to quickly narrow down the search, but these attributes only cover the range of addresses for the text associated with a compilation unit entry. To find the debugging information associated with a data object, an exhaustive search would be needed. Furthermore, any search through debugging information entries for different compilation units within a large program would potentially require the access of many memory pages, probably hurting debugger performance.

To make lookups of program objects by name or by address faster, a producer of DWARF information may provide two different types of tables containing information about the debugging information entries owned by a particular compilation unit entry in a more condensed format.

3.10.1 Lookup by Name

For lookup by name, a table is maintained in a separate object file section called `.debug_pubnames`. The table consists of sets of variable length entries, each set describing the names of global objects whose definitions or declarations are represented by debugging information entries owned by a single compilation unit. Each set begins with a header containing four values: the total length of the entries for that set, not including the length field itself, a version number, the offset from the beginning of the `.debug` section of the compilation unit entry referenced by the set and the size in bytes of the contents of the `.debug` section generated to represent that compilation unit. This header is followed by a variable number of offset/name pairs. Each pair consists of the offset from the beginning of the compilation unit entry corresponding to the current set of the debugging information entry for the given object, followed by a null-terminated character string representing the name of the object as given by the `AT_name` attribute of the referenced debugging entry. Each set of names is terminated by zero.

3.10.2 Lookup by Address

For lookup by address, a table is maintained in a separate object file section called `.debug_aranges`. The table consists of sets of variable length entries, each set describing the portion of the program's address space that is covered by a single compilation unit. Each set begins with a header containing three values: the total length of the entries for that set, not including the length field itself, a version number and the offset from the beginning of the `.debug` section of the compilation unit entry referenced by the set. This header is followed by a variable number of pairs of address range descriptors. Each pair consists of the beginning address of a range of text or data covered by some entry owned by the corresponding compilation unit entry, followed by the length of that range. A particular set is terminated by a pair consisting of two zeroes. By scanning the table, a debugger can quickly decide which compilation unit to look in to find the debugging information for an object that has a given address.

3.11 Line Number Table

A source-level debugger will need to know how to associate statements in the source files with the corresponding machine instruction addresses in the executable object or the shared objects used by that executable object. Such an association would make it possible for the debugger user to specify machine instruction addresses in terms of source statements. This would be done by specifying the line number in the source file containing the statement. The debugger can also use this information to display locations in terms of the source files and to single step from statement to statement.

As mentioned in section 3.1, above, the table of source statement information generated for a compilation unit currently exists in the `.line` section of an object file and is referenced by a corresponding compilation unit debugging information entry in the `.debug` section. The first entry in a source statement table contains the length of the table followed by the address of the first machine instruction generated for the compilation unit. The remainder of the table contains a list of source statement entries. A source statement entry contains a source line number, a statement position within the source line and an address delta. The line numbers in the source statement entries are numbered from the beginning of the compilation unit, starting with 1.

At the discretion of the compiler implementer, the value of the statement position in the source statement entry is either the number of characters from the beginning of the line to the beginning of the statement or a reserved, special value, `SOURCE_NO_POS`, to represent source information in terms of source lines alone.

The address delta in each source statement entry in a table of source statement entries is the address of the first machine instruction generated for that source statement minus the address of the first machine instruction generated for the compilation unit.

A single source statement extending over more than one source line has a source line information entry that refers to the line containing the start of that statement.

It is not necessary to have a source line information entry for every source line in a compilation unit, and there is no restriction on the order in which they appear.

The list of source line information entries is terminated by an entry whose line number is zero and whose delta represents the address of the first machine instruction beyond the last statement in the compilation unit.

This last entry allows a debugger to determine which instructions are associated with the last statement in the compilation unit.

4. DATA REPRESENTATION

This section describes the binary representation of the debugging information entry itself, of the attribute types and of other fundamental elements described above.

4.1 Vendor Extensibility

To reserve a portion of the DWARF name space and ranges of enumeration values for use for vendor specific extensions, special labels are reserved for tag names, attribute names, fundamental types, location atoms, type modifiers and language names. The labels denoting the beginning and end of the reserved value range for vendor specific extensions consist of the appropriate prefix (`TAG`, `AT`, `FT`, `OP`, `MOD` or `LANG`, respectively) followed by `_lo_user` or `_hi_user`. For example, for entry tags, the special labels are `TAG_lo_user` and `TAG_hi_user`. Values in the range between *prefix*`_lo_user` and *prefix*`_hi_user` inclusive, are reserved for vendor specific extensions. Vendors may use values in this range without conflicting with current or future system-defined values. All other values are reserved for use by the system.

Vendor defined tag, attribute, fundamental type, location atom, type modifier and language names conventionally use the form *prefix_vendor_id_name*, where *vendor_id* is some identifying character sequence chosen so as to avoid conflicts with other vendors.

To ensure that extensions added by one vendor may be safely ignored by consumers that do not understand those extensions, the following rules should be followed:

1. New attributes should be added in such a way that a debugger may recognize the format of a new attribute value without knowing the content of that attribute value.
2. The semantics of any new attributes should not alter the semantics of previously existing attributes.
3. The semantics of any new tags should not conflict with the semantics of previously existing tags.

4.2 Reserved Error Values

As a convenience for consumers of DWARF information, the value 0 is reserved in the encodings for attribute names, attribute forms, location atoms, fundamental types, type modifiers and languages to represent an error condition or unknown value. DWARF does not specify names for these reserved values, since they do not represent valid encodings for the given type and should not appear in DWARF debugging information.

4.3 Debugging Information Entry

Each debugging information entry consists of a 4-byte (inclusive) length followed by a 2-byte tag followed by a list of attributes. The tag encodings are given in Figure 8.

The 4-byte length is an unsigned integer whose value is the total number of bytes used by the debugging information entry. The value of the tag determines the classification of the debugging information entry.

On some architectures, there are alignment constraints on section boundaries. To make it easier to pad debugging information sections to satisfy such constraints, a debugging information entry with a length of less than eight (8) bytes is considered a null entry.

Tag name	Value
TAG_padding	0x0000
TAG_array_type	0x0001
TAG_class_type	0x0002
TAG_entry_point	0x0003
TAG_enumeration_type	0x0004
TAG_formal_parameter	0x0005
TAG_global_subroutine	0x0006
TAG_global_variable	0x0007
TAG_label	0x000a
TAG_lexical_block	0x000b
TAG_local_variable	0x000c
TAG_member	0x000d
TAG_pointer_type	0x000f
TAG_reference_type	0x0010
TAG_compile_unit	0x0011
TAG_source_file	0x0011
TAG_string_type	0x0012
TAG_structure_type	0x0013
TAG_subroutine	0x0014
TAG_subroutine_type	0x0015
TAG_typedef	0x0016
TAG_union_type	0x0017
TAG_unspecified_parameters	0x0018
TAG_variant	0x0019
TAG_common_block	0x001a
TAG_common_inclusion	0x001b
TAG_inheritance	0x001c
TAG_inlined_subroutine	0x001d
TAG_module	0x001e
TAG_ptr_to_member_type	0x001f
TAG_set_type	0x0020
TAG_subrange_type	0x0021
TAG_with_stmt	0x0022
TAG_lo_user	0x4080
TAG_hi_user	0xffff

Figure 8. Tag encodings

Null entries are different from entries with the special tag `TAG_padding` in that padding entries have a 4-byte size (whose value is greater than or equal to 8) and a 2-byte tag, followed by an appropriate number of unspecified padding bytes. Null entries consist of between 1 and 7 zero bytes.

4.4 Attribute Types

Attributes are represented by a 2-byte name field followed by an appropriate value. The form of the value is encoded into the attribute name. The possible forms (with their symbolic names) are:

address — Represented as an object of appropriate size to hold an address on the target machine (`FORM_ADDR`). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object.

reference	Represented as a 4-byte value (`FORM_REF`). The value is a byte offset relative to the beginning of the `.debug` section. It is relocated by the linkage editor.
constant	There are three forms of constants: A 2-byte data halfword (`FORM_DATA2`), a 4-byte data word (`FORM_DATA4`) and an 8-byte data doubleword (`FORM_DATA8`). These values may or may not be affected by linkage editing.
block	Blocks come in two forms. The first consists of a 2-byte length followed by 0 to 65,535 contiguous information bytes (`FORM_BLOCK2`). The second consists of a 4-byte length followed by 0 to 4,294,967,295 contiguous information bytes (`FORM_BLOCK4`). In both forms, the length is the number of information bytes that follow. The information bytes may contain any mixture of relocated (or relocatable) addresses, references to other debugging information entries or data bytes.
string	A block of contiguous non-null bytes followed by one null byte (`FORM_STRING`).

The form encodings are listed in Figure 9.

Form name	Value
FORM_ADDR	0x1
FORM_REF	0x2
FORM_BLOCK2	0x3
FORM_BLOCK4	0x4
FORM_DATA2	0x5
FORM_DATA4	0x6
FORM_DATA8	0x7
FORM_STRING	0x8

Figure 9. Attribute form encodings

The attribute encodings use the attribute form encodings just described. They are given in Figures 10 and 11.

4.5 Executable Objects and Shared Objects

The relocated addresses in the debugging information for an executable object are virtual addresses and the relocated addresses in the debugging information for a shared object are offsets relative to the start of the lowest segment used by that shared object.

This requirement makes the debugging information for shared objects position independent. Virtual addresses in a shared object may be calculated by adding the offset to the base address at which the object was attached. This offset is available in the run-time linker's data structures.

4.6 File Constraints

All debugging information entries in a relocatable object file, executable object or shared object are required to be physically contiguous.

Attribute name	Form	Value
AT_sibling	reference	(0x0010 | FORM_REF)
AT_location	block2	(0x0020 | FORM_BLOCK2)
AT_name	string	(0x0030 | FORM_STRING)
AT_fund_type	halfword	(0x0050 | FORM_DATA2)
AT_mod_fund_type	block2	(0x0060 | FORM_BLOCK2)
AT_user_def_type	reference	(0x0070 | FORM_REF)
AT_mod_u_d_type	block2	(0x0080 | FORM_BLOCK2)
AT_ordering	halfword	(0x0090 | FORM_DATA2)
AT_subscr_data	block2	(0x00a0 | FORM_BLOCK2)
AT_byte_size	word	(0x00b0 | FORM_DATA4)
AT_bit_offset	halfword	(0x00c0 | FORM_DATA2)
AT_bit_size	word	(0x00d0 | FORM_DATA4)
AT_element_list	block4	(0x00f0 | FORM_BLOCK4)
AT_stmt_list	word	(0x0100 | FORM_DATA4)
AT_low_pc	address	(0x0110 | FORM_ADDR)
AT_high_pc	address	(0x0120 | FORM_ADDR)
AT_language	word	(0x0130 | FORM_DATA4)
AT_member	reference	(0x0140 | FORM_REF)
AT_discr	reference	(0x0150 | FORM_REF)
AT_discr_value	block2	(0x0160 | FORM_BLOCK2)
AT_string_length	block2	(0x0190 | FORM_BLOCK2)
AT_common_reference	reference	(0x01a0 | FORM_REF)
AT_comp_dir	string	(0x01b0 | FORM_STRING)
AT_const_value	string	(0x01c0 | FORM_STRING)
AT_const_value	halfword	(0x01c0 | FORM_DATA2)
AT_const_value	word	(0x01c0 | FORM_DATA4)
AT_const_value	double word	(0x01c0 | FORM_DATA8)
AT_const_value	block2	(0x01c0 | FORM_BLOCK2)
AT_const_value	block4	(0x01c0 | FORM_BLOCK4)
AT_containing_type	reference	(0x01d0 | FORM_REF)
AT_default_value	address	(0x01e0 | FORM_ADDR)
AT_default_value	halfword	(0x01e0 | FORM_DATA2)
AT_default_value	double word	(0x01e0 | FORM_DATA8)
AT_default_value	string	(0x01e0 | FORM_STRING)
AT_friends	block2	(0x01f0 | FORM_BLOCK2)

Figure 10. Attribute encodings (part 1)

4.7 Location Atoms

Each atom in a location description has a one byte code that identifies that atom. The value of the identifying byte is interpreted to mean ‘‘reg(number),’’ ‘‘basereg(number),’’ ‘‘addr(address),’’ ‘‘const(number),’’ ‘‘deref2,’’ ‘‘deref,’’ or ‘‘add.’’ For all location atoms that require a (number), the identifying byte is followed by a value whose size is the same as the size of the fundamental type `FT_long` for the target machine. The (address) following an ‘‘addr’’ atom is of a size appropriate to represent an address on the target machine. All atoms in a location description are concatenated from left to right. The encoding of the atoms in a location description is described in Figure 12.

Attribute name	Form	Value
AT_inline	string	(0x0200|FORM_STRING)
AT_is_optional	string	(0x0210|FORM_STRING)
AT_lower_bound	reference	(0x0220|FORM_REF)
AT_lower_bound	halfword	(0x0220|FORM_DATA2)
AT_lower_bound	word	(0x0220|FORM_DATA4)
AT_lower_bound	double word	(0x0220|FORM_DATA8)
AT_program	string	(0x0230|FORM_STRING)
AT_private	string	(0x0240|FORM_STRING)
AT_producer	string	(0x0250|FORM_STRING)
AT_protected	string	(0x0260|FORM_STRING)
AT_prototyped	string	(0x0270|FORM_STRING)
AT_public	string	(0x0280|FORM_STRING)
AT_pure_virtual	string	(0x0290|FORM_STRING)
AT_return_addr	block2	(0x02a0|FORM_BLOCK2)
AT_specification	reference	(0x02b0|FORM_REF)
AT_start_scope	word	(0x02c0|FORM_DATA4)
AT_stride_size	word	(0x02e0|FORM_DATA4)
AT_upper_bound	reference	(0x02f0|FORM_REF)
AT_upper_bound	halfword	(0x02f0|FORM_DATA2)
AT_upper_bound	word	(0x02f0|FORM_DATA4)
AT_upper_bound	double word	(0x02f0|FORM_DATA8)
AT_virtual	string	(0x0300|FORM_STRING)
AT_lo_user	—	0x2000
AT_hi_user	—	0x3ff0

Figure 11. Attribute encodings (part 2)

Atom name	Value
OP_REG	0x01
OP_BASEREG	0x02
OP_ADDR	0x03
OP_CONST	0x04
OP_DEREF2	0x05
OP_DEREF	0x06
OP_DEREF4	0x06
OP_ADD	0x07
OP_lo_user	0xe0
OP_hi_user	0xff

Figure 12. Location atom encodings

A location description is the value of a location attribute and is stored in a block of contiguous bytes with a 2-byte length.

4.8 Fundamental Types

The encodings for the required fundamental type values are listed in Figure 13. For values in the range from `FT_lo_user` through `FT_hi_user`, inclusive, the low order byte of the fundamental type code contains the size in bytes of objects having the specified type, if the size is constant, otherwise the low order byte contains 0.

Type name	Value
FT_char	0x0001
FT_signed_char	0x0002
FT_unsigned_char	0x0003
FT_short	0x0004
FT_signed_short	0x0005
FT_unsigned_short	0x0006
FT_integer	0x0007
FT_signed_integer	0x0008
FT_unsigned_integer	0x0009
FT_long	0x000a
FT_signed_long	0x000b
FT_unsigned_long	0x000c
FT_pointer	0x000d
FT_float	0x000e
FT_dbl_prec_float	0x000f
FT_ext_prec_float	0x0010
FT_complex	0x0011
FT_dbl_prec_complex	0x0012
FT_void	0x0014
FT_boolean	0x0015
FT_ext_prec_complex	0x0016
FT_label	0x0017
FT_lo_user	0x8000
FT_hi_user	0xffff

Figure 13. Type encodings

4.9 Type Modifiers

Modifier types are represented by a single byte value. The encodings for the required values are given in Figure 14.

Modifier name	Value
MOD_pointer_to	0x01
MOD_reference_to	0x02
MOD_const	0x03
MOD_volatile	0x04
MOD_lo_user	0x80
MOD_hi_user	0xff

Figure 14. Type modifier encodings

4.10 Source Languages

Source languages are represented by a 4-byte constant. The encodings for the required values are given in Figure 15.

4.11 Friend Lists

The list of friends to a structure, union or class type that is the value of an AT_friends attribute is contained in a block of contiguous bytes with a 2-byte length. Each entry in the list is a 4-byte reference to another debugging information entry.

Language name	Value
LANG_C89	0x00000001
LANG_C	0x00000002
LANG_ADA83	0x00000003
LANG_C_PLUS_PLUS	0x00000004
LANG_COBOL74	0x00000005
LANG_COBOL85	0x00000006
LANG_FORTRAN77	0x00000007
LANG_FORTRAN90	0x00000008
LANG_PASCAL83	0x00000009
LANG_MODULA2	0x0000000a
LANG_lo_user	0x00008000
LANG_hi_user	0x0000ffff

Figure 15. Language encodings

4.12 Array Type Entries

4.12.1 Array Ordering

The encodings for the values of the order attributes of arrays is given in Figure 16.

Ordering name	Value
ORD_row_major	0
ORD_col_major	1

Figure 16. Ordering encodings

4.12.2 Array Subscripts

The components of a subscript data value are represented as follows:

Format specifier: 1-byte constant. The encodings are given in figure 17.

Fundamental type: 2-byte constant.

User-defined type: 4-byte reference.

Subscript bound index: Constant whose size is the same as the size of the fundamental type `FT_long` on the target machine.

Subscript bound location: Data block with a 2-byte length.

Element type: Fundamental type, user-defined type, modified fundamental type, or modified user-defined type, preceded by the corresponding 2-byte tag.

Note that the size of the complete subscript entry must be less than 65,536 bytes. A typical C array will require 11 bytes per dimension, plus the element type description (at least five bytes), allowing only 5,957 dimensions in an array type. Languages with dynamic array bounds will be restricted to even fewer than this number.

4.13 Enumeration Type Entries

Information about the enumeration literals is stored in an element list attribute whose value is a list of data elements stored in a block of contiguous bytes with a 4-byte length.

Format name	Value
FMT_FT_C_C	0x0
FMT_FT_C_X	0x1
FMT_FT_X_C	0x2
FMT_FT_X_X	0x3
FMT_UT_C_C	0x4
FMT_UT_C_X	0x5
FMT_UT_X_C	0x6
FMT_UT_X_X	0x7
FMT_ET	0x8

Figure 17. Format encodings

Each data item in an element list consists of a signed constant whose size is the same as the size of the fundamental type `FT_long` on the target machine, followed by a null-terminated block of contiguous bytes.

4.14 Name Lookup Table

Each set of entries in the table of global names contained in the `.debug_pubnames` section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 1-byte version identifier containing the value 1 for DWARF Version 1; a 4-byte offset into the `.debug` section; and a 4-byte length containing the size in bytes of the contents of the `.debug` section generated to represent this compilation unit. This header is followed by a series of tuples. Each tuple consists of a 4-byte offset followed by a string of non-null bytes terminated by one null byte. Each set is terminated by a 4-byte word containing the value 0.

4.15 Address Range Table

Each set of entries in the table of address ranges contained in the `.debug_aranges` section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 1-byte version identifier containing the value 1 for DWARF Version 1; and a 4-byte offset into the `.debug` section. This header is followed by a series of tuples. Each tuple consists of an address (in the size appropriate for the given architecture) and of a 4-byte constant length. Each set of tuples is terminated by a 0 for the address and 0 for the length.

4.16 Line Number Table

The table of source statement information generated for a compilation unit consists of a 4-byte length followed by a relocated address followed by a list of source statement entries. The 4-byte length is the total number of bytes occupied by the source statement information for the compilation unit, including the four bytes for the length. The relocated address is the address of the first machine instruction generated for that compilation unit.

A source statement entry contains a source line number (as an unsigned 4-byte integer), a statement position within the source line (as an unsigned 2-byte integer) and an address delta (as an unsigned 4-byte integer). The special statement position `SOURCE_NO_POS` has the value `0xffff`, and indicates that the statement entry refers to the entire source line.

4.17 Dependencies

The debugging information in this format is intended to exist in the `.debug`, `.debug_aranges`, `.debug_pubnames`, and `.line` sections of an object file. The information is not word-aligned, so the assembler must provide a way for the compiler to produce 2-byte and 4-byte quantities without alignment restrictions, and the linker must be able to relocate a 4-byte reference at an arbitrary alignment. In target architectures with 64-bit addresses, the assembler and linker must similarly handle 8-byte references at arbitrary alignments.

5. FUTURE DIRECTIONS

The UNIX International Programming Languages Special Interest Group is currently working to define Version 2 of the DWARF Debugging Information Format. This version will use a much denser encoding than does Version 1. It will also provide enhancements to the representation of statement information and locations, support for other features of the languages supported in DWARF Version 1 (such as macro information) and possibly support for other languages, as well. Information on Version 2 of DWARF can be obtained by contacting UNIX International.

Appendix 1 -- Current Attributes by Tag Value

The list below enumerates the attributes that are most applicable to each type of debugging information entry. DWARF does not in general require that a given debugging information entry contain a particular attribute or set of attributes. Instead, a DWARF producer is free to generate any, all, or none of the attributes described in the text as being applicable to a given entry. Other attributes (both those defined within this document but not explicitly associated with the entry in question, and new, vendor-defined ones) may also appear in a given debugging entry. Therefore, the list may be taken as instructive, but cannot be considered definitive.

For example, the list below states that an entry with a tag of `TAG_inheritance` may contain a type attribute, a location, a sibling, and the four ‘‘flag’’ attributes `AT_private`, `AT_protected`, `AT_public`, and `AT_virtual`. Obviously, not every `TAG_inheritance` entry will contain all of the above attributes; indeed, the first three of these ‘‘flag’’ attributes are mutually exclusive. Furthermore, in C++ the only ‘‘type’’ attribute which may appear is a user-defined type, since inheritance is not defined for fundamental or modified types. However, other languages with similar concepts may find a use for inheritance from fundamental types, or even modified fundamental or modified user-defined types. Thus we list all four possible ‘‘type’’ attributes as ‘‘applicable’’ to `TAG_inheritance`. A consumer need not be able to process a `TAG_inheritance` entry with an `AT_fund_type` attribute, if that combination is nonsensical in the language it understands, but it should gracefully ignore such a combination.

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_array_type	AT_byte_size AT_name AT_ordering AT_private AT_protected AT_public AT_sibling AT_start_scope AT_stride_size AT_subscr_data
TAG_class_type	AT_byte_size AT_friends AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope
TAG_common_block	AT_location AT_name AT_sibling
TAG_common_inclusion	AT_common_reference AT_sibling
TAG_compile_unit	AT_comp_dir AT_high_pc AT_language AT_low_pc AT_name AT_producer AT_sibling AT_stmt_list
TAG_entry_point	FT/MFT/UDT/MUDT† AT_low_pc AT_name AT_return_addr AT_sibling

† AT_fund_type, AT_mod_fund_type, AT_user_def_type or AT_mod_u_d_type.

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_enumeration_type	AT_byte_size AT_element_list AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope
TAG_formal_parameter	FT/MFT/UDT/MUDT AT_default_value AT_is_optional AT_location AT_name AT_sibling
TAG_global_subroutine	FT/MFT/UDT/MUDT AT_high_pc AT_inline AT_location AT_low_pc AT_member AT_name AT_private AT_program AT_protected AT_prototyped AT_public AT_pure_virtual AT_return_addr AT_start_scope AT_virtual AT_sibling
TAG_global_variable	FT/MFT/UDT/MUDT AT_constant_value AT_location AT_member AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_inheritance	AT_location AT_private AT_protected AT_public AT_sibling AT_user_def_type AT_virtual
TAG_inlined_subroutine	AT_high_pc AT_low_pc AT_sibling AT_specification
TAG_label	AT_low_pc AT_name AT_start_scope AT_sibling
TAG_lexical_block	AT_high_pc AT_low_pc AT_name AT_sibling
TAG_local_variable	FT/MFT/UDT/MUDT AT_constant_value AT_location AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope
TAG_member	FT/MFT/UDT/MUDT AT_byte_size AT_bit_offset AT_bit_size AT_location AT_name AT_private AT_protected AT_public AT_sibling

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_module	AT_high_pc AT_low_pc AT_name AT_private AT_protected AT_public AT_sibling
TAG_padding	
TAG_pointer_type	FT/MFT/UDT/MUDT AT_name AT_private AT_protected AT_public AT_start_scope AT_sibling
TAG_ptr_to_member_type	FT/MFT/UDT/MUDT AT_containing_type AT_name AT_sibling
TAG_reference_type	FT/MFT/UDT/MUDT AT_name AT_private AT_protected AT_public AT_start_scope AT_sibling
TAG_set_type	FT/MFT/UDT/MUDT AT_byte_size AT_name AT_private AT_protected AT_public AT_start_scope AT_sibling

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_string_type	AT_byte_size AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope AT_string_length
TAG_structure_type	AT_byte_size AT_friends AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope
TAG_subrange_type	FT/MFT/UDT/MUDT AT_byte_size AT_lower_bound AT_name AT_private AT_protected AT_public AT_sibling AT_upper_bound
TAG_subroutine	FT/MFT/UDT/MUDT AT_high_pc AT_inline AT_low_pc AT_member AT_name AT_private AT_protected AT_prototyped AT_public AT_return_addr AT_start_scope AT_sibling

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_subroutine_type	FT/MFT/UDT/MUDT AT_name AT_private AT_protected AT_prototyped AT_public AT_sibling AT_start_scope
TAG_typedef	FT/MFT/UDT/MUDT AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope
TAG_union_type	AT_byte_size AT_friends AT_name AT_private AT_protected AT_public AT_sibling AT_start_scope
TAG_unspecified_parameters	AT_sibling
TAG_variant	AT_discr AT_discr_value AT_location AT_name AT_private AT_protected AT_public AT_sibling
TAG_with_statement	AT_high_pc AT_location AT_low_pc AT_private AT_protected AT_public AT_sibling AT_user_def_type

Appendix 2 -- Example

The following is an example of a C program and a sample generation of DWARF information. It is provided for clarification and should not be construed as the ‘‘correct’’ or only way that a compiler may generate DWARF for the same program.

typedef.c:

```
typedef long LONG;
typedef char *POINTER;

POINTER p;
LONG l;

enum e { A, B, C };

struct a {
        int b;
        struct a *next;
};

typedef struct c {
        int d;
} cstruct;

typedef char BLOCK[1024];
typedef POINTER PAGE[66][80];

static BLOCK b;

int
myfunc(int i)
{
        BLOCK b;
        static struct a a1;
        struct a *aptr;
        cstruct c1;

        aptr = &a1;
        aptr->b = i;
        aptr->next = 0;
        return 1;
}

void foo()
{
}

static float bar()
{
        return 1.0;
}
```

Annotated contents of the `.debug` section for `typedef.o`, compiled with `-g`:

Each record is displayed in the following format:

hex_offset: <*decimal_length*> TAG_*tagname*
    AT_*attrname*(*value*)
    AT_*attrname*(*value*)

```
0000: <121> TAG_compile_unit
        AT_sibling(0x3a8)                      next compilation unit record
        AT_name(``typedef.c'')
        AT_producer(``Best Compiler Corp: C Compiler Version 1.3'')
        AT_comp_dir(``mymachine:/home/mydir/src'')
        AT_language(LANG_C89)
        AT_low_pc(0x0)                         will be relocated
        AT_high_pc(0x55)                       will be relocated
        AT_stmt_list(0x0)                      offset in .line section

0079: <23> TAG_typedef                         owned by compilation unit record
        AT_sibling(0x90)                       next record owned by my parent
        AT_name(``LONG'')
        AT_fund_type(FT_long)

0090: <29> TAG_typedef                         owned by compilation unit record
        AT_sibling(0xad)                       next record owned by my parent
        AT_name(``POINTER'')
        AT_mod_fund_type(<3>MOD_pointer_to FT_char)

00ad: <46> TAG_enumeration_type                owned by compilation unit record
        AT_sibling(0xdb)                       next record owned by my parent
        AT_name(``e'')
        AT_byte_size(0x4)
        AT_element_list(<18>(2=``C'') (1=``B'') (0=``A'') )

00db: <22> TAG_structure_type                  owned by compilation unit record
        AT_sibling(0x139)                      next record owned by my parent
        AT_name(``a'')
        AT_byte_size(0x8)

00f1: <30> TAG_member                          owned by struct ``a''
        AT_sibling(0x10f)                      next record owned by my parent
        AT_name(``b'')
        AT_fund_type(FT_integer)
        AT_location(<6>OP_CONST(0x0) OP_ADD )

010f: <38> TAG_member                          owned by struct ``a''
        AT_sibling(0x135)                      next record owned by my parent
        AT_name(``next'')
        AT_mod_user_def_type
              (<5>MOD_pointer_to 0xdb)         reference to record at offset 0xdb
        AT_location(<6>OP_CONST(0x4) OP_ADD)

0135: <4>                                      a null entry, end of sibling chain for struct ``a''
```

```
0139: <22> TAG_structure_type                owned by compilation unit record
        AT_sibling(0x171)                    next record owned by my parent
        AT_name(``c'')
        AT_byte_size(0x4)

014f: <30> TAG_member                        owned by struct ``c''
        AT_sibling(0x16d)                    next record owned by my parent
        AT_name(``d'')
        AT_fund_type(FT_integer)
        AT_location(<6>OP_CONST(0x0) OP_ADD )

016d: <4>                                    a null entry, end of sibling chain for struct ``c''

0171: <28> TAG_typedef                       owned by compilation unit record
        AT_sibling(0x18d)                    next record owned by my parent
        AT_name(``cstruct'')
        AT_user_def_type(0x139)              reference to record at offset 0139

018d: <36> TAG_array_type                    owned by compilation unit record
        AT_sibling(0x1b1)                    next record owned by my parent
        AT_ordering(0x0)
        AT_subscr_data(<16>FT_signed_integer[0:1023],
                FMT_ET: AT_fund_type(FT_char))

01b1: <26> TAG_typedef                       owned by compilation unit record
        AT_sibling(0x1cb)                    next record owned by my parent
        AT_name(``BLOCK'')
        AT_user_def_type(0x18d)              reference to record at offset 018d

01cb: <50> TAG_array_type                    owned by compilation unit record
        AT_sibling(0x1fd)                    next record owned by my parent
        AT_ordering(0x0)
        AT_subscr_data(<30>FT_signed_integer[0:65], FT_signed_integer[0:79],
                FMT_ET: AT_mod_fund_type(<3>MOD_pointer_to FT_char))

01fd: <25> TAG_typedef                       owned by compilation unit record
        AT_sibling(0x216)                    next record owned by my parent
        AT_name(``PAGE'')
        AT_user_def_type(0x1cb)              reference to record at offset 01cb
```

```
0216: <37> TAG_global_subroutine             owned by compilation unit record
        AT_sibling(0x2f8)                    next record owned by my parent
        AT_name(``myfunc'')
        AT_fund_type(FT_integer)
        AT_low_pc(0x0)                       will be relocated
        AT_high_pc(0x38)                     will be relocated

023b: <35> TAG_formal_parameter              owned by subroutine ``myfunc''
        AT_sibling(0x25e)                    next record owned by my parent
        AT_name(``i'')
        AT_fund_type(FT_integer)
        AT_location(<11>OP_BASEREG(0x5) OP_CONST(0x8) OP_ADD)

025e: <37> TAG_local_variable                owned by subroutine ``myfunc''
        AT_sibling(0x283)                    next record owned by my parent
        AT_name(``b'')
        AT_user_def_type(0x1b1)              reference to record at offset 01b1
        AT_location(<11>OP_BASEREG(0x5) OP_CONST(0xfffffc00) OP_ADD )

0283: <32> TAG_local_variable                owned by subroutine ``myfunc''
        AT_sibling(0x2a3)                    next record owned by my parent
        AT_name(``a1'')
        AT_user_def_type(0xdb)               reference to record at offset 00db
        AT_location(<5>OP_ADDR(0x0) )        will be relocated

02a3: <43> TAG_local_variable                owned by subroutine ``myfunc''
        AT_sibling(0x2ce)                    next record owned by my parent
        AT_name(``aptr'')
        AT_mod_user_def_type
             (<5>MOD_pointer_to 0xdb)        reference to record at offset 0xdb
        AT_location(<11>OP_BASEREG(0x5) OP_CONST(0xfffffbfc) OP_ADD)


02ce: <38> TAG_local_variable                owned by subroutine ``myfunc''
        AT_sibling(0x2f4)                    next record owned by my parent
        AT_name(``c1'')
        AT_user_def_type(0x139)               reference to record at offset 0139
        AT_location(<11>OP_BASEREG(0x9) OP_CONST(0x400) OP_ADD )

02f4: <4>                                    a null entry, end of sibling chain for ``myfunc''
```

```
02f8: <30> TAG_global_subroutine              owned by compilation unit record
        AT_sibling(0x31a)                     next record owned by my parent
        AT_name(``foo'')
        AT_low_pc(0x0)                        will be relocated
        AT_high_pc(0x41)                      will be relocated

0316: <4>                                     a null entry, end of sibling chain for ``foo''

031a: <34> TAG_subroutine                     owned by compilation unit record
        AT_sibling(0x340)                     next record owned by my parent
        AT_name(``bar'')
        AT_fund_type(FT_float)
        AT_low_pc(0x0)                        will be relocated
        AT_high_pc(0x55)                      will be relocated

033c: <4>                                     a null entry, end of sibling chain for ``bar''

0340: <31> TAG_local_variable                 owned by compilation unit record
        AT_sibling(0x35f)                     next record owned by my parent
        AT_name(``b'')
        AT_user_def_type(0x18d)               reference to record at offset 018d
        AT_location(<5>OP_ADDR(0x0) )         will be relocated

035f: <29> TAG_global_variable                owned by compilation unit record
        AT_sibling(0x37c)                     next record owned by my parent
        AT_name(``l'')
        AT_fund_type(FT_long)
        AT_location(<5>OP_ADDR(0x0) )         will be relocated

037c: <32> TAG_global_variable                owned by compilation unit record
        AT_sibling(0x39c)                     next record owned by my parent
        AT_name(``p'')
        AT_mod_fund_type(<3>MOD_pointer_to FT_char)
        AT_location(<5>OP_ADDR(0x0) )         will be relocated

039c: <4>                                     a null entry, end of sibling chain for ``typedef.c''
```

Contents of the `.line` section:

```
128              0
 25    -1        0
 31    -1        2
 32    -1        c
 33    -1       17
 34    -1       24
 35    -1       2b
 38    -1       38
 39    -1       3a
 42    -1       44
 43    -1       46
 44    -1       4e
  0    -1       55
```

length base address(will be relocated)
line# char offset hex address offset(not relocated)
line# == 0 is end of list

Table of Contents

List of Figures